1. 网络概述

网络编程技术是当前一种主流的编程技术，随着联网趋势的逐步增强以及网络应用程序的大量出现，所以在实际的开发中网络编程技术获得了大量的使用。

1.1、IP 地址和端口号

1.1.2 域名：

但是由于 IP 地址不容易记忆，所以为了方便记忆，有创造了另外一个概念——域名(Domain Name)，例如 sohu.com 等。一个 IP 地址可以对应多个域名，一个域名只能对应一个 IP 地址。

1.1.1 IP 地址：

为了能够方便的识别网络上的每个设备，网络中的每个设备都会有一个唯一的数字标识，这个就是 IP 地址。

IP4 协议：每个 IP 地址由 4 个 0-255 之间的数字组成，每个接入网络的计算机都拥有唯一的 IP 地址。

域名

1.1.3 DNS 服务器

在网络中传输的数据，全部是以 IP 地址作为地址标识，所以在实际传输数据以前需要将域名转换为 IP 地址，实现这种功能的服务器称

之为 DNS 服务器，也就是通俗的说法叫做域名解析。例如当用户在浏览器输入域名时，浏览器首先请求 DNS 服务器，将域名转换为 IP 地址，然后将转换后的 IP 地址反馈给浏览器，然后再进行实际的数据传输。

1.1.4 端口号：

在硬件上规定，端口的号码必须位于 0-65535 之间，每个端口唯一的对应一个网络程序，一个网络程序可以使用多个端口。这样一个网络程序运行在一台计算上时，不管是客户端还是服务器，都是至少占用一个端口进行网络通讯。在接收数据时，首先发送给对应的计算机，然后计算机根据端口把数据转发给对应的程序。

1.2、两种网络编程结构

1.2.1 C/S 结构

在网络通讯中，第一次主动发起通讯的程序被称作客户端(Client)程序，简称客户端，而在第一次通讯中等待连接的程序被称作服务器端(Server)程序，简称服务器。一旦通讯建立，则客户端和服务器端完全一样，没有本质的区别。

由此，网络编程中的两种程序就分别是客户端和服务器端，例如 QQ 程序，每个 QQ 用户安装的都是 QQ 客户端程序，而 QQ 服务器端程序则运行在腾讯公司的机房中，为大量的 QQ 用户提供服务。这种网络编程的结构被称作客户端/服务器结构，也叫做 Client/Server

结构，简称 C/S 结构。

使用 C/S 结 构的程序，在开发时需要分别开发客户端和服务器端，这种结构的优势在于由于客户端是专门开发的，所以根据需要实现各种效果，专业点说就是表现力丰富，而服 务器端也需要专门进行开发。但是这种结构也存在着很多不足，例如通用性差，几乎不能通用等，也就是说一种程序的客户端只能和对应的服务器端通讯，而不能和 其它服务器端通讯，在实际维护时，也需要维护专门的客户端和服务器端，维护的压力比较大。

1.2.2. B/S 结构

其实在运行很多程序时，没有必要使用专用的客户端，而需要使用通用的客户端，例如浏览器，使用浏览器作为客户端的结构被称作浏览器/服务器结构，也叫做 Browser/Server 结构，简称为 B/S 结构。

使用 B/S 结构的程序，在开发时只需要开发服务器端即可，这种结构的优势在于开发的压力比较小，不需要维护客户端。但是这种结构也存在着很多不足，例如浏览器的限制比较大，表现力不强，无法进行系统级操作等。

1.2.3 p2p 简介

P2P 程序是一种特殊的程序，应该一个 P2P 程序中既包含客户端程序，也包含服务器端程序，例如 BT，使用客户端程序部分连接其它的种子(服务器端)，而使用服务器端向其它的 BT 客户端传输数据。常

见的如 BT、电驴等。

1.3 协议

在实际进行数据交换时，为了让接收端理解该数据，计算机比较笨，什么都不懂的，那么就需要规定该数据的格式，这个数据的格式就是协议。在实际的网络程序编程中，最麻烦的内容不是数据的发送和接收，因为这个功能在几乎所有的程序语言中都提供了封装好的 API 进行调用，最麻烦的内容就是协议的设计以及协议的生产和解析，这个才是网络编程中最核心的内容。

1.4 两种网络通讯方式

1.4.2 UDP(用户数据报协议)方式

UDP 方式就类似于发送短信，使用这种方式进行网络通讯时，不需要建立专门的虚拟连接，传输也不是很可靠，如果发送失败则客户端无法获得。

1.4.1 TCP(传输控制协议)方式

在网络通讯中，TCP 方式就类似于拨打电话，使用该种方式进行网络通讯时，需要建立专门的虚拟连接，然后进行可靠的数据传输，如果数据发送失败，则客户端会自动重发该数据。

2. 网络编程步骤

2.1、客户端编程

1、 建立网络连接

客户端网络编程的第一步都是建立网络连接。在建立网络连接时需要指定连接到的服务器的 IP 地址和端口号，建立完成以后，会形成一条虚拟的连接，后续的操作就可以通过该连接实现数据交换了。

2、 交换数据

连接建立以后，就可以通过这个连接交换数据了。交换数据严格按照请求响应模型进行，由客户端发送一个请求数据到服务器，服务器反馈一个响应数据给客户端，如果客户端不发送请求则服务器端就不响应。根据逻辑需要，可以多次交换数据，但是还是必须遵循请求响应模型。

3、 关闭网络连接

在数据交换完成以后，关闭网络连接，释放程序占用的端口、内存等系统资源，结束网络编程。

最基本的步骤一般都是这三个步骤，在实际实现时，步骤 2 会出现重复，在进行代码组织时，由于网络编程是比较耗时的操作，所以一般开启专门的现场进行网络通讯。

2.2、 服务器端编程

服务器端(Server)是指在网络编程中被动等待连接的程序，服务器端一般实现程序的核心逻辑以及数据存储等核心功能。服务器端的编程步骤和客户端不同，是由四个步骤实现，依次是：

1、 监听端口

服务器端属于被动等待连接，所以服务器端启动以后，不需要发起连接，而只需要监听本地计算机的某个固定端口即可。这个端口就是服务器端开放给客户端的端口，服务器端程序运行的本地计算机的 IP 地址就是服务器端程序的 IP 地址。

2、 获得连接

当客户端连接到服务器端时，服务器端就可以获得一个连接，这个连接包含客户端的信息，例如客户端 IP 地址等等，服务器端和客户端也通过该连接进行数据交换。

一般在服务器端编程中，当获得连接时，需要开启专门的线程处理该连接，每个连接都由独立的线程实现。

3、 交换数据

服务器端通过获得的连接进行数据交换。服务器端的数据交换步骤是首先接收客户端发送过来的数据，然后进行逻辑处理，再把处理以后的结果数据发送给客户端。简单来说，就是先接收再发送，这个和客

户端的数据交换数序不同。

其实，服务器端获得的连接和客户端连接是一样的，只是数据交换的步骤不同。当然，服务器端的数据交换也是可以多次进行的。在数据交换完成以后，关闭和客户端的连接。

4、 关闭连接

当服务器程序关闭时，需要关闭服务器端，通过关闭服务器端使得服务器监听的端口以及占用的内存可以释放出来，实现了连接的关闭。

其实服务器端编程的模型和呼叫中心的实现是类似的，例如移动的客服电话 10086 就是典型的呼叫中心，当一个用户拨打 10086 时，转接给一个专门的客服人员，由该客服实现和该用户的问题解决，当另外一个用户拨打 10086 时，则转接给另一个客服，实现问题解决，依次类推。

在服务器端编程时，10086 这个电话号码就类似于服务器端的端口号码，每个用户就相当于一个客户端程序，每个客服人员就相当于服务器端启动的专门和客户端连接的线程，每个线程都是独立进行交互的。

这就是服务器端编程的模型，只是 TCP 方式是需要建立连接的，对于服务器端的压力比较大，而 UDP 是不需要建立连接的，对于服务器端的压力比较小罢了。

3. 基础的网络类——InetAddress 类

该类的功能是代表一个 IP 地址，并且将 IP 地址和域名相关的操作方法包含在该类的内部。

在示例代码中，演示了 InetAddress 类的基本使用，并使用了该类中的几个常用方法，该代码的执行结果是：

www.163.com/220.181.28.50

/127.0.0.1

super 瑞/192.168.1.100

域名：super 瑞

IP:192.168.1.100

说明：由于该代码中包含一个互联网的网址，所以运行该程序时需要联网，否则将产生异常。

在后续的使用中，经常包含需要使用 InetAddress 对象代表 IP 地址的构造方法，当然，该类的使用不是必须的，也可以使用字符串来代表 IP 地址进行实现。

```
public class InetAddressDemo {
    public static void main(String[] args) {
        try {
            // 使用域名创建对象
            InetAddress inet1 = InetAddress.getByName("www.163.com");
            System.out.println(inet1);
            // 使用 IP 创建对象
            InetAddress inet2 = InetAddress.getByName("127.0.0.1");
            System.out.println(inet2);
            // 获得本机地址对象
```

```
            InetAddress inet3 = InetAddress.getLocalHost();
            System.out.println(inet3);

            // 获得对象中存储的域名

            String host = inet3.getHostName();

            System.o・ut.println("域名： " + host);


            // 获得对象中存储的 IP

            String ip = inet3.getHostAddress();
            System.out.println("IP:" + ip);
        } catch (Exception e) {
        }
    }
}
```

4. TCP 编程

在 Java 语言中，对于 TCP 方式的网络编程提供了良好的支持，在实际实现时，以 java.net.Socket 类代表客户端连接，以 java.net.ServerSocket 类代表服务器端连接。在进行网络编程时，底层网络通讯的细节已经实现了比较高的封装，所以在程序员实际编程时，只需要指定 IP 地址和端口号码就可以建立连接了。正是由于这种高度的封装，一方面简化了 Java 语言网络编程的难度，另外也使得使用 Java 语言进行网络编程时无法深入到网络的底层，所以使用 Java 语言进行网络底层系统编程很困难，具体点说，Java 语言无法实现底层的网络嗅探以及获得 IP 包结构等信息。但是由于 Java 语言的网络编程比较简单，所以还是获得了广泛的使用。

4.1 客户端

1. 建立连接

在客户端网络编程中，首先需要建立连接，在 Java API 中以 java.net.Socket 类的对象代表网络连接，所以建立客户端网络连接，也就是创建 Socket 类型的对象，该对象代表网络连接，示例如下：

```
Socket socket1 = new Socket( “192.168.1.103” ,10000);
Socket socket2 = new Socket( “www.sohu.com” ,80);
```

上面的代码中，socket1 实现的是连接到 IP 地址是 192.168.1.103 的计算机的 10000 号端口，而 socket2 实现的是连接到域名是 www.sohu.com 的计算机的 80 号端口，至于底层网络如何实现建立连接，对于程序员来说是完全透明的。如果建立连接时，本机网络不通，或服务器端程序未开启，则会抛出异常。

2. 交换数据

连接一旦建立，则完成了客户端编程的第一步，紧接着的步骤就是按照“请求-响应”模型进行网络数据交换，在 Java 语言中，数据传输功能由 Java IO 实现，也就是说只需要从连接中获得输入流和输出流即可，然后将需要发送的数据写入连接对象的输出流中，在发送完成以后从输入流中读取数据即可。示例代码如下：

```
OutputStream os = socket1.getOutputStream(); //获得输出流
```

InputStream is = socket1.getInputStream();　　//获得输入流

上面的代码中，分别从 socket1 这个连接对象获得了输出流和输入流对象，在整个网络编程中，后续的数据交换就变成了 IO 操作，也就是遵循“请求-响应”模型的规定，先向输出流中写入数据，这些数据会被系统发送出去，然后在从输入流中读取服务器端的反馈信息，这样就完成了一次数据交换过程，当然这个数据交换过程可以多次进行。

这里获得的只是最基本的输出流和输入流对象，还可以根据前面学习到的 IO 知识，使用流的嵌套将这些获得到的基本流对象转换成需要的装饰流对象，从而方便数据的操作。

3. 关闭连接

最后当数据交换完成以后，关闭网络连接，释放网络连接占用的系统端口和内存等资源，完成网络操作，示例代码如下：

```
socket1.close();
```

这就是最基本的网络编程功能介绍。下面是一个简单的网络客户端程序示例，该程序的作用是向服务器端发送一个字符串“Hello”，并将服务器端的反馈显示到控制台，数据交换只进行一次，当数据交换进行完成以后关闭网络连接，程序结束。

代码示例：

```
public class MyClient {
   public static void main(String[] args) {
      // 1.建立客户端连接，创建socket对象
```

```
        InetAddress inetAddress = null;
        Socket socket = null;
        int port = 15000;
        try {
            inetAddress = InetAddress.getLocalHost();
            socket = new Socket(inetAddress, port);
        } catch (Exception e) {
            e.printStackTrace();
        }
        // 2.交换数据
        InputStream is = null;
        OutputStream os = null;
        String data = "hello,我是客户端";
        try {
            os = socket.getOutputStream(); // 通过IO方式发送数据
            os.write(data.getBytes());
            System.out.println("客户端(我)说: " + data);
            is = socket.getInputStream();
            byte b[] = new byte[1024];
            is.read(b);
            System.out.println("服务器端回答道: " + new
String(b));
        } catch (IOException e) {
            e.printStackTrace();
        } // 3.关闭连接
        finally {
            try {
                os.close();
                is.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

4.2 服务器端

1. 监听端口号

在服务器端程序编程中，由于服务器端实现的是被动等待连

接，所以服务器端编程的第一个步骤是监听端口，也就是监听是否有客户端连接到达。实现服务器端监听的代码为：

```
ServerSocket ss = new ServerSocket(10000);
```

该代码实现的功能是监听当前计算机的 10000 号端口，如果在执行该代码时，10000 号端口已经被别的程序占用，那么将抛出异常。否则将实现监听。

2. 获得连接

服务器端编程的第二个步骤是获得连接。该步骤的作用是当有客户端连接到达时，建立一个和客户端连接对应的 Socket 连接对象，从而释放客户端连接对于服务器端端口的占用。实现功能就像公司的前台一样，当一个客户到达公司时，会告诉前台我找某某某，然后前台就通知某某某，然后就可以继续接待其它客户了。通过获得连接，使得客户端的连接在服务器端获得了保持，另外使得服务器端的端口释放出来，可以继续等待其它的客户端连接。实现获得连接的代码是：

```
Socket socket = ss.accept();
```

该代码实现的功能是获得当前连接到服务器端的客户端连接。需要说明的是 accept 和前面 IO 部分介绍的 read 方法一样，都是一个阻塞方法，也就是当无连接时，该方法将阻塞程序的执行，直到连接到达时才执行该行代码。另外获得

的连接会在服务器端的该端口注册，这样以后就可以通过在服务器端的注册信息直接通信，而注册以后服务器端的端口就被释放出来，又可以继续接受其它的连接了。

3. 数据交换

连接获得以后，后续的编程就和客户端的网络编程类似了，这里获得的 Socket 类型的连接就和客户端的网络连接一样了，只是服务器端需要首先读取发送过来的数据，然后进行逻辑处理以后再发送给客户端，也就是交换数据的顺序和客户端交换数据的步骤刚好相反。

4. 关闭连接

```
ss.close();
```

代码示例：

```
public class MyServer {
    public static void main(String[] args) {
        // 1.创建监听
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(15000);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 2.获得连接
        Socket socket = null;
        try {
            socket = serverSocket.accept();
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
        // 3. 交换数据
        String response = "收到消息，你好，这是服务器回应";
        InputStream is = null;
        OutputStream os = null;
        try {
            is = socket.getInputStream();
            byte b[] = new byte[1024];
            is.read(b);
            System.out.println("客户端对我说：" + new String(b));
            os = socket.getOutputStream();
            os.write(response.getBytes());
            System.out.println("我是服务器端，我反馈说：" +
response);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 4.关闭连接
        try {
            os.close();
            is.close();
            socket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4.3 复用 socket 连接

建立连接以后，将数据交换的逻辑写到一个循环中就可以了。这样只要循环不结束则连接就不会被关闭。按照这种思路，可以改造一下上面的代码，让该程序可以在建立连接一次以后，发送三次数据，当然这里的次数也可以是多次。

4.4 如何使服务器支持多个客户端同时工作？

一个服务器端一般都需要同时为多个客户端提供通讯如果需要同时支持多个客户端，则必须使用前面介绍的线程的概念。简单来说，也就是当服务器端接收到一个连接时，启动一个专门的线程处理和该客户端的通讯。

按照这个思路改写的服务端示例程序将由两个部分组成，MulThreadSocketServer 类实现服务器端控制，实现接收客户端连接，然后开启专门的逻辑线程处理该连接，LogicThread 类实现对于一个客户端连接的逻辑处理，将处理的逻辑放置在该类的 run 方法中。

logicThread 中不能进行流的关闭操作，否则 socket 对象会被关闭

示例代码：

```
public class MulThreadSocketServer {
   public static void main(String[] args) {
      ServerSocket serverSocket = null;
      Socket socket = null;
      // 1.监听端口号
      try {
         serverSocket = new ServerSocket(15000);
         System.out.println("服务器已启动监听！");
      } catch (IOException e) {
         e.printStackTrace();
      }
      // 2.获得连接
      while (true) {
         try {
            socket = serverSocket.accept();
            // 3.启动线程进行数据交换
            new LogicThread(socket);
         } catch (IOException e) {
            e.printStackTrace();
         }// 4. 关闭连接
         finally {
```

```
            try {
                socket.close();
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class LogicThread extends Thread {
    public Socket socket;
    public InputStream is;
    public OutputStream os;

    public LogicThread(Socket socket) {
        this.socket = socket;
        start(); // 启动线程
    }

    public void run() {
        byte[] b = new byte[1024];
        try {
            // 初始化流
            os = socket.getOutputStream();
            is = socket.getInputStream();
            // 读取数据
            int n = is.read(b);
            System.out.println("客户端发来请求" + new String(b));
            // 逻辑处理
            byte[] response = response(b, n);
            // 反馈数据
            os.write(response);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            close();
        }
    }

    /**
     * 关闭流和连接
     */
    private void close() {
```

```
        try {
            // 关闭流和连接
            os.close();
            is.close();
            socket.close();
        } catch (Exception e) {
        }
    }
/**
     * 根据客户端请求实现响应
     *
     * @param b
     * @param length
     * @return
     */
    private byte[] response(byte[] b, int length) {
        // 根据客户端请求作出响应
        String data = "服务器响应：我做出了一次响应";
        byte[] response = data.getBytes();
        System.out.println(data);
        return response;
    }
}
```

在该示例代码中，实现了一个 while 形式的死循环，由于 accept 方法是阻塞方法，所以当客户端连接未到达时，将阻塞该程序的执行，当客户端到达时接收该连接，并启动一个新的 LogicThread 线程处理该连接，然后按照循环的执行流程，继续等待下一个客户端连接。这样当任何一个客户端连接到达时，都开启一个专门的线程处理，通过多个线程支持多个客户端同时处理。

在该示例代码中，每次使用一个连接对象构造该线程，该连接对象就是该线程需要处理的连接，在线程构造完成以后，该线程就被启动起来了，然后在 run 方法内部对客户端连接进行处理，数据交换的逻辑和前面的示例代码一致，只是这里将接收到客户端发送过来的数

据并进行处理的逻辑封装成了 logic 方法，按照前面介绍的 IO 编程的内容，客户端发送过来的内容存储在数组 b 的起始下标为 0，长度为 n 个中，这些数据是客户端发送过来的有效数据，将有效的数据传递给 logic 方法，logic 方法实现的是 echo 服务的逻辑，也就是将客户端发送的有效数据形成以后新的 response 数组，并作为返回值反馈。

在线程中将 logic 方法的返回值反馈给客户端，这样就完成了服务器端的逻辑处理模拟，其他的实现和前面的介绍类似，这里就不在重复了。

这里的示例还只是基础的服务器端实现，在实际的服务器端实现中，由于硬件和端口数的限制，所以不能无限制的创建线程对象，而且频繁的创建线程对象效率也比较低，所以程序中都实现了线程池来提高程序的执行效率。

这里简单介绍一下线程池的概念，线程池(Thread pool)是池技术的一种，就是在程序启动时首先把需要个数的线程对象创建好，例如创建 5000 个线程对象，然后当客户端连接到达时从池中取出一个已经创建完成的线程对象使用即可。当客户端连接关闭以后，将该线程对象重新放入到线程池中供其它的客户端重复使用，这样可以提高程序的执行速度，优化程序对于内存的占用等。

5. UDP 编程

UDP(User Datagram Protocol)，中文意思是用户数据报协议，方式

类似于发短信息，是一种物美价廉的通讯方式，使用该种方式无需建立专用的虚拟连接，由于无需建立专用的连接，所以对于服务器的压力要比 TCP 小很多，所以也是一种常见的网络编程方式。但是使用该种方式最大的不足是传输不可靠，当然也不是说经常丢失，就像大家发短信息一样，理论上存在收不到的可能，这种可能性可能是 1%，反正比较小，但是由于这种可能的存在，所以平时我们都觉得重要的事情还是打个电话吧(类似 TCP 方式)，一般的事情才发短信息(类似 UDP 方式)。网络编程中也是这样，必须要求可靠传输的信息一般使用 TCP 方式实现，一般的数据才使用 UDP 方式实现。

在 Java API 中，实现 UDP 方式的编程，包含客户端网络编程和服务器端网络编程，主要由两个类实现，分别是：

DatagramSocket

DatagramSocket 类实现“网络连接”，包括客户端网络连接和服务器端网络连接。虽然 UDP 方式的网络通讯不需要建立专用的网络连接，但是毕竟还是需要发送和接收数据，DatagramSocket 实现的就是发送数据肘的发射器，以及接收数据肘的监听器的角色。类比于 TCP 中的网络连接，该类既可以用于实现客户端连接，也可以用于实现服务器端连接。

DatagramPacket

DatagramPacket 类实现对于网络中传输的数据封装，也就是说，该类的对象代表网络中交换的数据。在 UDP 方式的网络编程中，无论

是需要发送的数据还是需要接收的数据，都必须被处理成DatagramPacket类型的对象，该对象中包含发送到的地址、发送到的端口号以及发送的内容等。其实DatagramPacket类的作用类似于现实中的信件，在信件中包含信件发送到的地址以及接收人，还有发送的内容等，邮局只需要按照地址传递即可。在接收数据时，接收到的数据也必须被处理成DatagramPacket类型的对象，在该对象中包含发送方的地址、端口号等信息，也包含数据的内容。和TCP方式的网络传输相比，IO编程在UDP方式的网络编程中变得不是必须的内容，结构也要比TCP方式的网络编程简单一些。

5.1 客户端

5.1.1. 建立连接

UDP方式的建立连接和TCP方式不同，只需要建立一个连接对象即可，不需要指定服务器的IP和端口号码。实现的代码为：

```
DatagramSocket ds = new DatagramSocket();
```

这样就建立了一个客户端连接，该客户端连接使用系统随机分配的一个本地计算机的未用端口号。在该连接中，不指定服务器端的IP和端口，所以UDP方式的网络连接更像一个发射器，而不是一个具体的连接。

当然，可以通过制定连接使用的端口号来创建客户端连接。

```
DatagramSocket ds = new DatagramSocket(5000);
```

这样就是使用本地计算机的5000号端口建立了一个连接。一

般在建立客户端连接时没有必要指定端口号码。

5.1.2 发送数据与接收数据

1.发送数据

在 UDP 方式的网络编程中，IO 技术不是必须的，在发送数据时，需要将需要发送的数据内容首先转换为 byte 数组，然后将数据内容、服务器 IP 和服务器端口号一起构造成一个 DatagramPacket 类型的对象，这样数据的准备就完成了，发送时调用网络连接对象中的 send 方法发送该对象即可。

2.接收数据

UDP 方式在进行网络通讯时，也遵循“请求-响应”模型，在发送数据完成以后，就可以接收服务器端的反馈数据了。

下

面介绍一下 UDP 客户端编程中接收数据的实现。当数据发送出去以后，就可以接收服务器端的反馈信息了。接收数据在 Java 语言中的实现是这样的：首先构造一个数据缓冲数组，该数组用于存储接收的服务器端反馈数据，该数组的长度必须大于或等于服务器端反馈的实际有效数据的长度。然后以该缓冲数组为基础构造一个 DatagramPacket 数据包对象，最后调用连接对象的 receive 方法接收数据即可。接收到的服务器端反馈数据存储在 DatagramPacket 类型的对象内部。

5.1.3 关闭连接

UDP 方式客户端网络编程的最后一个步骤就是关闭连接。虽然 UDP 方式不建立专用的虚拟连接，但是连接对象还是需要占用系统资源，所以在使用完成以后必须关闭连接。关闭连接使用连接对象中的 close 方法即可，实现的代码如下：

ds.close();

需要说明的是，和 TCP 建立连接的方式不同，UDP 方式的同一个网络连接对象，可以发送到达不同服务器端 IP 或端口的数据包，这点是 TCP 方式无法做到的。

代码示例：

```
public class MyClient {
    public static void main(String[] args) {
        DatagramSocket ds = null;
        DatagramPacket dp = null;
        // 1.建立连接，创建DatagramSocket对象
        try {
            ds = new DatagramSocket();
        } catch (SocketException e) {
            e.printStackTrace();
        }
        String data = "hello,服务器！";
        byte[] buf = new byte[1024];
        buf = data.getBytes();
        InetAddress ia = null;
        try {
            ia = InetAddress.getLocalHost();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
        // 2.发送数据
        dp = new DatagramPacket(buf, buf.length, ia, 15000);
        try {
```

```
            ds.send(dp);
            System.out.println("客户端(我)对服务器说: " + data);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 接收数据
        byte[] ref = new byte[1024];
        DatagramPacket dpr = new DatagramPacket(ref,
ref.length);
        try {
            ds.receive(dpr);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 3.关闭连接
        ds.close();
    }
}
```

5.2 服务器端

UDP 方式网络编程的服务器端实现和 TCP 方式的服务器端实现类似，也是服务器端监听某个端口，然后获得数据包，进行逻辑处理以后将处理以后的结果反馈给客户端，最后关闭网络连接。

5.2.1 建立连接，监听端口号

首先 UDP 方式服务器端网络编程需要建立一个连接，该连接监听某个端口，实现的代码为：

```
DatagramSocket ds = new DatagramSocket(10010);
```

由于服务器端的端口需要固定，所以一般在建立服务器端连接时，都指定端口号。例如该示例代码中指定 10010 端口为服务器端使用的端口号，客户端端在连接服务器端时连接该端口号即可。

5.2.2 发送、接收数据

接着服务器端就开始接收客户端发送过来的数据，其接收的方法和客户端接收的方法一直，其中 receive 方法的作用类似于 TCP 方式中 accept 方法的作用，该方法也是一个阻塞方法，其作用是接收数据。接收到客户端发送过来的数据以后，服务器端对该数据进行逻辑处理，然后将处理以后的结果再发送给客户端，在这里发送时就比客户端要麻烦一些，因为服务器端需要获得客户端的 IP 和客户端使用的端口号，这个都可以从接收到的数据包中获得。示例代码如下：

```
//获得客户端的 IP
InetAddress clientIP = receiveDp.getAddress()
 //获得客户端的端口号
Int clientPort = receiveDp.getPort();
```

使用以上代码，就可以从接收到的数据包对象 receiveDp 中获得客户端的 IP 地址和客户端的端口号，这样就可以在服务器端中将处理以后的数据构造成数据包对象，然后将处理以后的数据内容反馈给客户端了。

5.2.3 关闭连接

当服务器端实现完成以后，关闭服务器端连接，实现的方式为调用连接对象的 close 方法，示例代码如下：

```
ds.close();
```

代码示例：

```
public class MyServer {
    public static void main(String[] args) {
        DatagramSocket ds = null;
        DatagramPacket dpr = null;
        // 1.监听端口号，建立连接
        try {
            ds = new DatagramSocket(15000);
            System.out.println("服务器已启动！");
        } catch (SocketException e) {
            e.printStackTrace();
        }
        // 2.接受数据
        byte[] buf = new byte[1024];
        dpr = new DatagramPacket(buf, buf.length);
        try {
            ds.receive(dpr);
            System.out.println("客户端对我说：" + new String(buf,
0, buf.length));
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 发送数据
        // 首先要获得服务器端IP和端口号
        int cport = dpr.getPort();
        InetAddress cip = dpr.getAddress();
        String data = "你好！ 我已收到来自客户端消息，这是服务器端响
应！";
        byte[] sef = new byte[1024];
        sef = data.getBytes();
        DatagramPacket dps = new DatagramPacket(sef, sef.length,
cip, cport);
        try {
            ds.send(dps);
            System.out.println("服务器（我）响应：" + data);
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 3.关闭连接
        ds.close();
    }
}
```

6. 线程池技术

6.1 简介

线程的使用在 java 中占有极其重要的地位，在 jdk1.4 极其之前的 jdk 版本中，关于线程池的使用是极其简陋的。在 jdk1.5 之后这一情况有了很大的改观。Jdk1.5 之后加入了 java.util.concurrent 包，这个包中主要介绍 java 中线程以及线程池的使用。为我们在开发中处理线程的问题提供了非常大的帮助。

6.2 线程池的作用

线程池作用就是限制系统中执行线程的数量。

根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了浪费了系统资源，多了造成系统拥挤效率不高。用线程池控制线程数量，其他线程排队等候。一个任务执行完毕，再从队列的中取最前面的任务开始执行。若队列中没有等待进程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程池中有等待的工作线程，就可以开始运行了；否则进入等待队列。

6.3 为什么要使用线程池

1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2.可以根据系统的承受能力，调整线程池中工作线线程的数目，防止

因为消耗过多的内存，而把服务器累趴下(每个线程需要大约 1MB 内存，线程开的越多，消耗的内存也就越大，最后死机)。

Java 里面线程池的顶级接口是 Executor，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。

比较重要的几个类：

ExecutorService	真正的线程池接口。
ScheduledExecutorService	能和 Timer/TimerTask 类似，解决那些需要任务重复执行的问题。
ThreadPoolExecutor	ExecutorService 的默认实现。
ScheduledThreadPoolExecutor	继承 ThreadPoolExecutor 的 ScheduledExecutorService 接口实现，周期性任务调度的类实现。

6.4 ThreadGroup 与 ThreadPoolExecutor

线程组表示一个线程的集合。此外，线程组也可以包含其他线程组。线程组构成一棵树，在树中，除了初始线程组外，每个线程组都有一个父线程组。允许线程访问 有关自己的线程组的信息，但是不允许它访问有关其线程组的父线程组或其他任何线程组的信息；线程消耗包括内存和其它系统资源在内的大量资源。除了 Thread 对象所需的内存之外，每个线程都需要两个可能很大的执行调用堆栈。除此以外，

JVM 可能会为每个 Java 线程创建一个本机线程，这些本机线程将消耗额外的系统资源。最后，虽然线程之间切换的调度开销很小，但如果有很多线程，环境切换也可能严重地影响程序的性能。

6.5 Executor 详解

Java 里面线程池的顶级接口是 Executor，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。ThreadPoolExecutor 是 Executors 类的底层实现。我们先介绍下 Executors。

java.util.concurrent.Executors 类，这个类提供大量创建连接池的静态方法。

6.5.1 固定大小线程池

代码示例：

```
public class MyFixedThreadPool {
    public static void main(String[] args) {
        // 创建一个可重用固定线程数的线程池
        int num = 2;
        ExecutorService esp = Executors.newFixedThreadPool(num);
        // 创建线程
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        // 将线程放入线程池
        esp.execute(t1);
        esp.execute(t2);
        esp.execute(t3);
        esp.execute(t4);
        esp.execute(t5);
```

```
        esp.shutdown();
    }
}
```

运行结果：

```
<terminated> MyFixedThreadPool [Java Application]
pool-1-thread-1线程正在执行。。。。
pool-1-thread-2线程正在执行。。。。
pool-1-thread-1线程正在执行。。。。
pool-1-thread-2线程正在执行。。。。
pool-1-thread-1线程正在执行。。。。
```

6.5.2 单任务线程池

在上例的基础上改一行创建 pool 对象的代码为：

//创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。

```
ExecutorService pool = Executors.newSingleThreadExecutor();
```

运行结果：

```
pool-1-thread-1线程正在执行。。。。
pool-1-thread-1线程正在执行。。。。
pool-1-thread-1线程正在执行。。。。
pool-1-thread-1线程正在执行。。。。
```

6.5.3 可变尺寸的线程池

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。

```
ExecutorService pool = Executors.newCachedThreadPool();
```

运行结果：

```
pool-1-thread-1线程正在执行。。。。
pool-1-thread-2线程正在执行。。。。
pool-1-thread-3线程正在执行。。。。
pool-1-thread-4线程正在执行。。。。
pool-1-thread-5线程正在执行。。。。
```

6.5.4 延迟连接池

代码示例：

```
public class MySchedulesThread {
    public static void main(String[] args) {
        // 创建一个线程池，它可安排在给定延迟后运行命令或者定期地执
行。
        ScheduledExecutorService pool =
Executors.newScheduledThreadPool(2);
        // 创建线程
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        Thread t4 = new MyThread();
        Thread t5 = new MyThread();
        // 将线程放入池中进行执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        // 使用延迟执行风格的方法
        pool.schedule(t4, 10, TimeUnit.SECONDS);
        pool.schedule(t5, 10, TimeUnit.SECONDS);
        // 关闭线程池
        pool.shutdown();
    }
}
```

6.6 ThreadPoolExecutor 详解

构造方法参数：

corePoolSize： 线程池维护线程的最少数量

maximumPoolSize：线程池维护线程的最大数量

keepAliveTime： 线程池维护线程所允许的空闲时间

unit： 线程池维护线程所允许的空闲时间的单位

workQueue： 线程池所使用的缓冲队列

handler：线程池对拒绝任务的处理策略 ThreadPoolExecutor 是 Executors 类的底层实现。

当一个任务通过 execute(Runnable)方法欲添加到线程池时 ：

如果此时线程池中的数量小于 corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务。

如果此时线程池中的数量等于 corePoolSize，但是缓冲队列 workQueue 未满，那么任务被放入缓冲队列。

如果此时线程池中的数量大于 corePoolSize，缓冲队列 workQueue 满，并且线程池中的数量小于 maximumPoolSize，建新的线程来处理被添加的任务。

如果此时线程池中的数量大于 corePoolSize，缓冲队列 workQueue 满，并且线程池中的数量等于 maximumPoolSize，那么通过 handler 所指定的策略来处理此任务。

也就是：处理任务的优先级为：

核心线程 corePoolSize、任务队列 workQueue、最大线程 maximumPoolSize，如果三者都满了，使用 handler 处理被拒绝的任务。

当线程池中的线程数量大于 corePoolSize 时，如果某线程空闲时间超过 keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数。

6.6.1 BlockingQueue

1.LinkedBlockingQueue：一个基于已链接节点的、范围任意的 blocking queue。此队列按 FIFO（先进先出）排序元素。队列的头部 是在队列中时间最长的元素。队列的尾部 是在队列中时间最短的元素。新元素插入到队列的尾部，并且队列获取操作会获得位于队列头部的元素。链接队列的吞吐量通常要高于基于数组的队列，但是在大多数并发应用程序中，其可预知的性能要低。

2.SynchronousQueue：一种阻塞队列，其中每个插入操作必须等待另一个线程的对应移除操作 ，反之亦然。

3.PriorityBlockingQueue：一个无界阻塞队列，它使用与类 PriorityQueue 相同的顺序规则，并且提供了阻塞获取操作。虽然此队列逻辑上是无界的，但是资源被耗尽时试图执行 add 操作也将失败（导致 OutOfMemoryError）。

所有 BlockingQueue 都可用于传输和保持提交的任务。可以使用此队列与池大小进行交互：

如果运行的线程少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。（如果当前运行的线程小于 corePoolSize，则任务根本不会存放，添加到 queue 中，而是直接抄家伙（thread）开始运行）

如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。

6.6.2 排队有三种通用策略：

直接提交。工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

无界队列。使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web 页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

有界队列。当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O 边界），则系统可能为超过您许可的更多线程安排时间。

使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

6.6.3 BlockingQueue 的选择。

例子一：使用直接提交策略，也即 SynchronousQueue。

首先 SynchronousQueue 是无界的，也就是说他存数任务的能力是没有限制的，但是由于该 Queue 本身的特性，在某次添加元素后必须等待其他线程取走后才能继续添加。在这里不是核心线程便是新创建的线程，但是我们试想一样下，下面的场景。

我们使用一下参数构造

ThreadPoolExecutor：

```
new ThreadPoolExecutor( 2, 3, 30, TimeUnit.SECONDS,
newSynchronousQueue<Runnable>(),
new RecorderThreadFactory("CookieRecorderPool"),
new ThreadPoolExecutor.CallerRunsPolicy());
```

当核心线程已经有 2 个正在运行.

1.此时继续来了一个任务（A），根据前面介绍的“如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。”,所以 A 被添加到 queue 中。

2.又来了一个任务（B），且核心 2 个线程还没有忙完，OK，接下来首先尝试 1 中描述，但是由于使用的 SynchronousQueue，所以一定无法加入进去。

3.此时便满足了上面提到的“如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。”，所以必然会新建一个线程来运行这个任务。

4.暂时还可以，但是如果这三个任务都还没完成，连续来了两个任务，第一个添加入 queue 中，后一个呢？queue 中无法插入，而线程数达到了 maximumPoolSize，所以只好执行异常策略了。

所以在使用 SynchronousQueue 通常要求 maximumPoolSize 是无界的，这样就可以避免上述情况发生（如果希望限制就直接使用有界队列）。对于使用 SynchronousQueue 的作用 jdk 中写的很清楚：此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。

什么意思？如果你的任务 A1，A2 有内部关联，A1 需要先运行，那么先提交 A1，再提交 A2，当使用 SynchronousQueue 我们可以保证，A1 必定先被执行，在 A1 么有被执行前，A2 不可能添加入 queue 中。

例子二：使用无界队列策略，即 LinkedBlockingQueue

这个就拿 newFixedThreadPool 来说，根据前文提到的规则：

如果运行的线程少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。那么当任务继续增加，会发生什么呢？

如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。OK，此时任务变加入队列之中了，那什么时候才会添加新线程呢？

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出

maximumPoolSize，在这种情况下，任务将被拒绝。这里就很有意思了，可能会出现无法加入队列吗？不像 SynchronousQueue 那样有其自身的特点，对于无界队列来说，总是可以加入的（资源耗尽，当然另当别论）。换句说，永远也不会触发产生新的线程！corePoolSize 大小的线程数会一直运行，忙完当前的，就从队列中拿任务开始运行。所以要防止任务疯长，比如任务运行的实行比较长，而添加任务的速度远远超过处理任务的时间，而且还不断增加，不一会儿就爆了。

例子三：有界队列，使用 ArrayBlockingQueue。

这个是最为复杂的使用，所以 JDK 不推荐使用也有些道理。与上面的相比，最大的特点便是可以防止资源耗尽的情况发生。

举例来说，请看如下构造方法：

```
 new ThreadPoolExecutor(2, 4, 30, TimeUnit.SECONDS,
new ArrayBlockingQueue<Runnable>(2),
new RecorderThreadFactory("CookieRecorderPool"),
new ThreadPoolExecutor.CallerRunsPolicy());
```

假设，所有的任务都永远无法执行完。

对于首先来的 A,B 来说直接运行，接下来，如果来了 C,D，他们会被放到 queue 中，如果接下来再来 E,F，则增加线程运行 E，F。但是如果再来任务，队列无法再接受了，线程数也到达最大的限制了，所以就会使用拒绝策略来处理。

6.6.4 keepAliveTime

jdk 中的解释是：当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。

有点拗口，其实这个不难理解，在使用了“池”的应用中，大多都有类似的参数需要配置。比如数据库连接池，DBCP 中的 maxIdle，minIdle 参数。

什么意思？接着上面的解释，后来向老板派来的工人始终是“借来的”，俗话说“有借就有还”，但这里的问题就是什么时候还了，如果借来的工人刚完成一个任务就还回去，后来发现任务还有，那岂不是又要去借？这一来一往，老板肯定头也大死了。

合理的策略：既然借了，那就多借一会儿。直到“某一段”时间后，发现再也用不到这些工人时，便可以还回去了。这里的某一段时间便是 keepAliveTime 的含义，TimeUnit 为 keepAliveTime 值的度量。

7. 线程池实现 Socket 复用

创建线程池可以通过调用 java.util.concurrent.Executors 类里的静态方法 newChahedThreadPool 或是 newFixedThreadPool 来创建，也可以通过新建一个 java.util.concurrent.ThreadPoolExecutor 实例来执行任务。这里我们采用 newFixedThreadPool 方法来建立线程池。

代码示例：

```
public class MyThreadPoolSocket {
   public static void main(String[] args) {
      // 创建线程池
```

```
        ExecutorService pool = Executors.newFixedThreadPool(2);
        ServerSocket serverSocket = null;
        Socket socket = null;
        // 监听端口号
        try {
            serverSocket = new ServerSocket(15000);
            System.out.println("服务器已启动监听！");
        } catch (IOException e) {
            e.printStackTrace();
        }
        int count = 0;
        try {
            while (true) {
                // 当有新连接建立时，accept返回时，将服务任务提交给线
程池执行。
                count++;
                socket = serverSocket.accept();
                System.out.println("第" + count + "次启动线程");
                Thread t = new LogicThread(socket);
                pool.execute(t);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                socket.close();
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

        }
    }
}
public class LogicThread extends Thread {
    public static int count = 0;
    public Socket socket;
    public InputStream is;
    public OutputStream os;
    private static ReentrantLock lock = new ReentrantLock();

    private int getCount() {
        int ret = 0;
        try {
```

```
            lock.lock();
            ret = count;
        } finally {
            lock.unlock();
        }
        return ret;
    }

    private void increaseCount() {
        try {
            lock.lock();
            ++count;
        } finally {
            lock.unlock();
        }
    }

    public LogicThread(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        byte[] b = new byte[1024];
        try {
            // 初始化流
            os = socket.getOutputStream();
            is = socket.getInputStream();
            // 读取数据
            int n = is.read(b);
            increaseCount();
            System.out.println(Thread.currentThread().getName() +
"线程正在第"
                    + getCount() + "次服务。。。");
            System.out.println("客户端发来请求: " + new
String(b));
            // 逻辑处理
            byte[] response = response(b, n);
            // 反馈数据
            os.write(response);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                is.close();
```

```
                os.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     * 根据客户端请求实现响应
     *
     * @param b
     * @param length
     * @return
     */
    private byte[] response(byte[] b, int length) {
        // 根据客户端请求作出响应
        String data = "服务器响应：我做出了一次响应";
        byte[] response = data.getBytes();
        System.out.println(data);
        return response;
    }
}
```

7.1 重入锁 ReentrantLock

线程维护一个 count 来记录服务线程被调用的次数。每当服务任务被调用一次时，count 的值自增 1，因此 ServiceThread 提供一个 increaseCount 和 getCount 的方法，分别将 count 值自增 1 和取得该 count 值。由于可能多个 线程存在竞争，同时访问 count，因此需要加锁机制，在 Java 5 之前，我们只能使用 synchronized 来锁定。Java 5 中引入了性能更加粒度更细的重入锁 ReentrantLock。我们使用 ReentrantLock 保证代码线程安全。

8. 网络协议

网络协议是指对于网络中传输的数据格式的规定。对于网络编程初学者来说，没有必要深入了解 TCP/IP 协议簇，所以对于初学者来说去读大部头的《TCP/IP 协议》也不是一件很合适的事情，因为深入了解 TCP/IP 协议是网络编程提高阶段，也是深入网络编程底层时才需要做的事情。

对于一般的网络编程来说，更多的是关心网络上传输的逻辑数据内容，也就是更多的是应用层上的网络协议

7.1 概念

网络协议的实质也是客户端程序和服务器端程序对于数据的一种约定，只是由于以计算机为基础，所以更多的是使用数字来代表内容，这样就显得比较抽象一些。

举一个简单的例子，介绍一些基础的网络协议设计的知识。例如需要设计一个简单的网络程序：网络计算器。也就是在客户端输入需要计算的数字和运算符，在 服务器端实现计算，并将计算的结果反馈给客户端。在这个例子中，就需要约定两个数据格式：客户端发送给服务器端的数据格式，以及服务器端反馈给客户端的数据格式。

可能你觉得这个比较简单，例如客户端输入的数字依次是 12 和 432，输入的运算符是加号，可能最容易想到的数据格式是形成字符串“12+432”，这样格式的确比较容易阅读，但是服务器端在进行计算时，逻辑就比较麻烦，因为需要首先拆分该字符串，然后才能进

行计算，所以可用的数据格式就有了一下几种：

“12，432，+” “格式为：第一个数字，第二个数字，运算符”

“12，+，432” “格式为：第一个数字，运算符，第二个数字”

其实以上两种数据格式很接近，比较容易阅读，在服务器端收到该数据格式以后，使用“，”为分隔符分割字符串即可。

假设对于运算符再进行一次约定，例如约定数字 0 代表+，1 代表减，2 代表乘，3 代表除，整体格式遵循以上第一种格式，则上面的数字生产的协议数据为：“12，432，0”

这就是一种基本的发送的协议约定了。

另外一个需要设计的协议格式就是服务器端反馈的数据格式，其实服务器端主要反馈计算结果，但是在实际接受数据时，有可能存在格式错误的情况，这样就需要简单的设计一下服务器端反馈的数据格式了。例如规定，如果发送的数据格式正确，则反馈结果，否则反馈字符串“错误”。这样就有了以下的数据格式：

客户端：“1,111,1” 服务器端：”-110”

客户端：“123，23，0” 服务器端：“146”

客户端：“1，2，5” 服务器端：“错误”

这样就设计出了一种最最基本的网络协议格式，从该示例中可以看出，网络协议就是一种格式上的约定，可以根据逻辑的需要约定出各种数据格式，在进行设计时一般遵循“简单、通用、容易解析”的原则进行。

而对于复杂的网络程序来说，需要传输的数据种类和数据量都比

较大，这样只需要依次设计出每种情况下的数据格式即可，例如 QQ 程序，在该程序中需要进行传输的网络数据种类很多，那么在设计时就可以遵循：登录格式、注册格式、发送消息格式等等，一一进行设计即可。所以对于复杂的网络程序来说，只是增加了更多的命令格式，在实际设计时的工作量增加不是太大。

不管怎么说，在网络编程中，对于同一个网络程序来说，一般都会涉及到两个网络协议格式：客户端发送数据格式和服务器端反馈数据格式，在实际设计时，需要一一对应。这就是最基本的网络协议的知识。

7.2 编码步骤

网络协议设计完成以后，在进行网络编程时，就需要根据设计好的协议格式，在程序中进行对应的编码了，客户端程序和服务器端程序需要进行协议处理的代码分别如下。

客户端程序需要完成的处理为：

1、 客户端发送协议格式的生成

2、 服务器端反馈数据格式的解析

服务器端程序需要完成的处理为：

1、 服务器端反馈协议格式的生成

2、 客户端发送协议格式的解析

这里的生成是指将计算好的数据，转换成规定的数据格式，这里的解析指，从反馈的数据格式中拆分出需要的数据。在进行对应的代码编

写时，严格遵循协议约定即可。

所以，对于程序员来说，在进行网络程序编写时，需要首先根据逻辑的需要设计网络协议格式，然后遵循协议格式约定进行协议生成和解析代码的编写，最后使用网络编程技术实现整个网络编程的功能。

由于各种网络程序使用不同的协议格式，所以不同网络程序的客户端之间无法通用。

而对于常见协议的格式，例如 HTTP(Hyper Text Transfer Protocol，超文本传输协议)、FTP(File Transfer Protocol，文件传输协议)，SMTP(Simple Mail Transfer Protocol，简单邮件传输协议)等等，都有通用的规定，具体可以查阅相关的 RFC 文档。

最后，对于一种网络程序来说，网络协议格式是该程序最核心的技术秘密，因为一旦协议格式泄漏，则任何一个人都可以根据该格式进行客户端的编写，这样将影响服务器端的实现，也容易出现一些其它的影响。

9. 网络编程实例

该示例实现的功能是质数判断，程序实现的功能为客户端程序接收用户输入的数字，然后将用户输入的内容发送给服务器端，服务器端判断客户端发送的数字是否是质数，并将判断的结果反馈给客户端，客户端根据服务器端的反馈显示判断结果。

质数的规则是：最小的质数是 2，只能被 1 和自身整除的自然数。当用户输入小于 2 的数字，以及输入的内容不是自然数时，都属于非法

输入。

网络程序的功能都分为客户端程序和服务器端程序实现，下面先描述一下每个程序分别实现的功能：

1、 客户端程序功能：

a)接收用户控制台输入

b)判断输入内容是否合法

c)按照协议格式生成发送数据

d)发送数据

e)接收服务器端反馈

f)解析服务器端反馈信息，并输出

2、 服务器端程序功能：

a)接收客户端发送数据

b)按照协议格式解析数据

c)判断数字是否是质数

d)根据判断结果，生成协议数据

e)将数据反馈给客户端

分解好了网络程序的功能以后，就可以设计网络协议格式了，如果该程序的功能比较简单，所以设计出的协议格式也不复杂。

客户端发送协议格式：

将用户输入的数字简单加密（数字 1 变 a,2 变 b…），再将字符串转换为 byte 数组即可。

客户端发送“quit”字符串代表结束连接

服务器端发送协议格式：

反馈数据长度为 1 个字节。数字 0 代表是质数，1 代表不是质数，2 代表协议格式错误。

例如客户端发送数字 12，则反馈 1，发送 13 则反馈 0，发送 0 则反馈 2。

功能设计完成以后，就可以分别进行客户端和服务器端程序的编写了，在编写完成以后联合起来进行调试即可。

9.1 TCP 实现

9.1.1 客户端发送数据协议

```
public class MyEncode {
    private MyEncode() {
    }

    public static int decode(String str) {
        str = str.replace("a", "1");
        str = str.replace("b", "2");
        str = str.replace("c", "3");
        str = str.replace("d", "4");
        str = str.replace("e", "5");
        str = str.replace("f", "6");
        str = str.replace("g", "7");
        str = str.replace("h", "8");
        str = str.replace("i", "9");
        str = str.trim();
        return Integer.parseInt(str);
    }

    public static String encode(int num) {
        String str = Integer.toString(num);
        str = str.replace("1", "a");
        str = str.replace("2", "b");
        str = str.replace("3", "c");
        str = str.replace("4", "d");
```

```
        str = str.replace("5", "e");
        str = str.replace("6", "f");
        str = str.replace("7", "g");
        str = str.replace("8", "h");
        str = str.replace("9", "i");
        return str;
    }
}
```

9.1.2 客户端

```
public class MyClient {
    public static void main(String[] args) {
        Socket socket = null;
        OutputStream os = null;
        InputStream is = null;
        // 1.创建连接
        try {
            socket = new Socket(InetAddress.getLocalHost(),
15000);
            os = socket.getOutputStream();
            is = socket.getInputStream();
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 2.控制台输入数据
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("请输入一个数字或\"quit\"来退出客户
端");
            String input = scanner.nextLine();
            byte[] data = new byte[1024];
            // 3.判断是否结束
            if ("quit".equals(input)) { // 结束交互
                data = input.getBytes();
                try {
                    os.write(data);
                } catch (IOException e) {
                    e.printStackTrace();
                }
                break;
            } else { // 按照协议生成数据
                input = MyEncode.encode(Integer.parseInt(input));
```

```
                data = input.getBytes();
                try { // 发送数据
                    os.write(data);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            // 接受服务器反馈
            byte[] buf = new byte[1024];
            try {
                is.read(buf);
                // 解析服务器数据并输出
                String res = new String(buf).trim();
                if ("1".equals(res)) {
                    System.out.println("服务器说不是质数");
                } else if ("0".equals(res)) {
                    System.out.println("服务器说是质数");
                } else if ("2".equals(res)) {
                    System.out.println("协议格式错误");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        try {
            is.close();
            os.close();
            socket.close();
            System.out.println("客户端已停止。。。");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9.1.3 服务器端

```
public class MyServer {
    public static void main(String[] args) {
        ServerSocket ss = null;
        Socket socket = null;
        OutputStream os = null;
        InputStream is = null;
```

```
        // 监听端口
        try {
            ss = new ServerSocket(15000);
            // 创建连接
            socket = ss.accept();
            System.out.println("服务器已启动！");
            os = socket.getOutputStream();
            is = socket.getInputStream();
        } catch (IOException e) {
            e.printStackTrace();
        }
        while (true) {
            byte[] re = new byte[1024];
            // 获取数据
            try {
                is.read(re);
            } catch (IOException e) {
                e.printStackTrace();
            }
            String data = new String(re).trim();
            if ("quit".equals(data)) { // 退出
                break;
            } else {
                // 按照格式解析数据
                int num = MyEncode.decode(data);
                // 判断是否是质数
                String response = "0"; // 默认不是质数
                for (int i = 2; i <= num / 2; i++) {
                    if (num % i == 0) {
                        response = "1";
                    }
                }
                if (num < 2) {
                    response = "2"; // 数据不合法
                }
                byte[] buf = new byte[1024];
                buf = response.getBytes();
                try {
                    os.write(buf);
                } catch (IOException e) {
                    e.printStackTrace();
                } // 发送数据
            }
        }
```

```
        // 关闭连接
        try {
            os.close();
            is.close();
            socket.close();
            System.out.println("服务器已停止...");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```